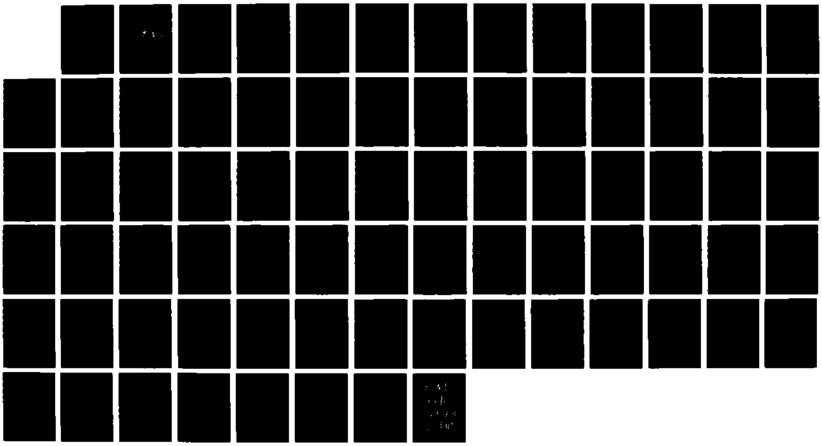
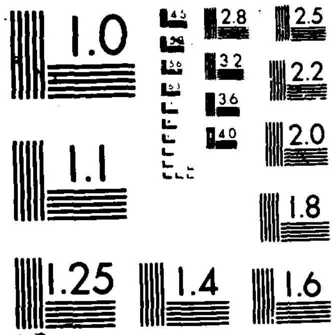
AD-A187 080   EXPERIENCE WITH OMEGA: IMPLEMENTATION OF A PROTOTYPE   1/1
PROGRAMMING ENVIRONMENT PART 6(U) NAVAL POSTGRADUATE
SCHOOL MONTEREY CA B J MACLENNAN SEP 87
UNCLASSIFIED NPS52-87-042-PT-6   F/G 12/5   NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

2

NPS52-87-042

NAVAL POSTGRADUATE SCHOOL

Monterey, California

EXPERIENCE WITH Ω

IMPLEMENTATION OF A PROTOTYPE PROGRAMMING ENVIRONMENT
PART VI

Bruce J. MacLennan

September 1987

Approved for public release; distribution unlimited

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Support for this research was provided by the Office of Naval Research under contract N00014-86-WR-24092. Preparation of this report was supported by ONR under contract N00014-87-WR-24037.

This report prepared by:

[illegible]
BRUCE J. MACLENNAN
Professor
of Computer Science

Reviewed by:

[illegible]
VINCENT Y. LUM
Chairman
Department of Computer Science

Released by:

[illegible]
JAMES M. FREMGEN
Acting Dean of Information and
Policy Sciences

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY Unclassified	3 DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-87-042	5 MONITORING ORGANIZATION REPORT NUMBER(S)

6a NAME OF PERFORMING ORGANIZATION Computer Science Department	6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943		7b ADDRESS (City, State, and ZIP Code) San Diego, CA
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N0001486WR24092

8c. ADDRESS (City, State, and ZIP Code)

10 SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
61153N	RR0150801	4331519-01	

11. TITLE (Include Security Classification)

Experiment with Ω Implementation of a Prototype Programming Environment Part VI

12. PERSONAL AUTHOR(S)
Bruce J. MacLennan

13a. TYPE OF REPORT Technical	13b TIME COVERED FROM ______ TO ______	14 DATE OF REPORT (Year, Month, Day) September 1987	15 PAGE COUNT 75

16. SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This is the last report of a series exploring the use of the Ω programming notation to prototype a programming environment. This environment includes an interpreter, unparser, syntax directed editor, command interpreter, debugger and code generator, and supports programming in a small applicative language. This report presents a universal (i.e., table-driven) syntax directed editor and unparser, which requires only 53 rules to express. A running implementation of these ideas is listed in the appendices.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION	
22a. NAME OF RESPONSIBLE INDIVIDUAL Bruce J. MacLennan	22b TELEPHONE (Include Area Code) (408) 646-2449	22c OFFICE SYMBOL 52M1

EXPERIENCE WITH Ω

IMPLEMENTATION OF A PROTOTYPE PROGRAMMING ENVIRONMENT

PART VI

Bruce J. MacLennan*
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

A-1

Abstract:

This is the last report of a series exploring the use of the Ω programming notation to prototype a programming environment. This environment includes an interpreter, unparser, syntax directed editor, command interpreter, debugger and code generator, and supports programming in a small applicative language. This report presents a universal (i.e., table-driven) syntax directed editor and unparser, which requires only 53 rules to express. A running implementation of these ideas is listed in the appendices.

I. Goals

A. Introduction

Our goal in this series of reports† [MacLennan85b, MacLennan85c, MacLennan86a, MacLennan86b, MacLennan86c] is to explore in the context of a very simple language the use of the Ω programming notation [MacLennan83, MacLennan85a] to implement some of the tools that constitute a programming environment.

In Part VI we address the following problem: Each extension to the language has required new rules to be added to the unparser and editor. However, the new rules have been very similar in every case. Therefore, our goal will be to make the unparser and editor *table-driven* (like a table-driven parser) so that only the tables need be changed when the language is extended.

What information needs to be provided to permit table-driven syntax-directed editing? We will certainly need the sort of syntactic information normally provided in a BNF grammar. This tells how pro-

* Author's current address: Department of Computer Science, Ayres Hall, University of Tennessee at Knoxville, Knoxville, Tennessee, 37996-1301.

† Support for this research was provided by the **Office of Naval Research** under contract N00014-86-WR-24092. Preparation of this report was supported by ONR under contract N00014-87-WR-24037.

gram trees are unparsed onto the display. An example of the sort of information required is shown in Table 1, which contains the grammar for our sample language; we will take it as typical of the kinds of grammars we would like to handle.

TABLE 1. Grammar for Sample Language

<program>	= <expr>
<expr>	= <arex> <rel> <arex>
	| <arex>
<rel>	= < | > | ≤ | ≥ | = | ≠
<arex>	= <arex> { + | − } <term>
	| <term>
<term>	= <term> { × | ÷ } <factor>
	| <factor>
<factor>	= <number>
	| <var>
	| <var> <factor>
	| (<expr>)
	| (**if** <expr> **then** <expr> **else** <expr>)
	| [**let** <var> = <expr> <expr>]
	| [**func** <var> <var> = <expr> <expr>]

B. Editing Commands

If we are to automatically generate a syntax-directed editor then we also need to know the selector-keys associated with the various language constructs. Table 2 shows an example of this information for the case of the sample language.

These selector commands are *language-dependent*, i.e., they depend on the kinds of constructs included in the language (although we would hope for some consistency in their choice). A number of other commands are *language-independent*: they are the same for all languages. For example, we have the navigation commands:

TABLE 2. Selector Keys for Sample Language

=	→	<arex> = <arex>
n	→	<arex> ≠ <arex>
<	→	<arex> < <arex>
>	→	<arex> > <arex>
[	→	<arex> ⩽ <arex>
]	→	<arex> ⩾ <arex>
+	→	<arex> + <term>
−	→	<arex> − <term>
*	→	<term> × <factor>
/	→	<term> ÷ <factor>
#	→	<number>
	→	<var>
c	→	<var> <factor>
(	→	(<expr>)
i	→	(**if** <expr> **then** <expr> **else** <expr>)
l	→	|**let** <var> = <expr> <expr> |
f	→	|**func** <var> <var> = <expr> <expr> |

in, out, next, prev, root

We also have the **delete** command, which works on any language construct (the insertion commands are language-dependent; they are the selector keys). Finally, we will postulate a single mechanism for entering strings of characters, whether for identifiers, literal constants, comments, or whatever:

$$\_c_1c_2c_3 \cdots c_n@$$

The '_' key (typically the spacebar) initiates the accumulation of characters, and the '@' key (typically the carriage return) completes the accumulation and enters the string into the structure.

C. Example Session

Suppose we wish to enter the following program:

```
let K = 16
 func fac n =
  (if n = 0
    then 1
    else n × fac (n − 1) )
  fac (K ÷ 2) | |
```

When the system is entered we will be prompted for a program:

<program>

Typing the 'l' selector key calls for the creation of a let block[1]:

l

[**let** *<var>* = <expr>

<expr>]

The bound variable is entered, and we move to the bound value:

_K@ **next**

[**let** K = *<expr>*

<expr>]

The literal constant '16' is entered, and we move to the body:

#_16@ **next**

[**let** K = 16

<expr>]

The 'f' key calls for the creation of a function definition block:

f

[**let** K = 16

[**func** *<var>* <var> = <expr>

<expr>]]

The function name is entered:

_fac@ **next**

func fac *<var>* = <expr>

<expr>

The formal parameter is entered:

_n@ **next**

[**func** fac n = *<expr>*

1. In each case we show the command typed, followed by the resulting display.

<expr>]

An **if** expression is requested:

i

[**func** fac n =

(**if** *<expr>*

then <expr>

else <expr>)

<expr>]

For the expression 'n = 0', we first enter the operator:

=

<arex> = <arex>

Next we enter the variable 'n' and move right:

_n@ **next**

n = *<arex>*

The condition is completed by entering the number '0':

_0@

n = 0

Moving right focuses attention on the consequent of the conditional:

next

(**if** n = 0

then *<expr>*

else <expr>)

The literal constant '1' is entered as the consequent of the **if**:

_1@

1

The next component is the alternate:

next

(**if** n = 0

then 1

else <*expr*>)

The product of n and the factorial of n−1 is entered as the alternate. First we request a product:

*

<*term*> × <factor>

We enter the variable 'n' and move right:

_n@ **next**

n × <*factor*>

The 'c' key requests a function call:

c

n × <*var*> <factor>

We enter 'fac' and move right:

_fac@ **next**

n × fac <*factor*>

For the argument we request a parenthesized expression:

(

n × fac (<*expr*>)

Notice that the closing parenthesis is automatically supplied, so there is no possibility of misnested parentheses. The '−' key requests a difference:

−

n × fac (<*arex*> − <term>)

The difference is completed routinely:

_n@ **next** # _1@

```
    n × fac (n − 1)
```

Moving to the next component brings us to the body of the function definition block:

```
  next
    [let K = 16
      [func fac n =
        (if ...
          then ...
          else ... )
        <expr> ] ]
```

We have shown the components of the conditional as '...', representing suppressed detail, although on a reasonably sized screen the entire program would be shown. (This suppression of detail is called *holo-phrasting* in the literature.) The body of the function definition block is the invocation of factorial with k÷2:

```
  c _fac@ next / _K@ next # _2@
    fac (K ÷ 2)
```

We shift our focus back to the root, where we see the completed program:

```
  root
    [let K = 16
      [func fac n =
        (if n = 0
          then 1
          else n × fac (n − 1) )
        fac (K ÷ 2) ] ]
```

D. Transaction Grammar

How might we express concisely the connection between concrete syntax, abstract syntax and editing commands? We will use a translation grammar augmented by selector keys, which we call a *transaction grammar*. There are a variety of notations for transaction grammars. For example:

```
<arex>:

  + → <arex> + <term>

    ⇒ Sum (Left = <arex>, Right = <term>)

  - → <arex> - <term>

    ⇒ Dif (Left = <arex>, Right = <term>)

  else <term>
```

This means that if a '+' key is typed, then a Sum node is generated (which is unparsed '<arex> − <term>'). The **else** clause means that if any key other that '+' or '−' is typed when <arex> is expected, then that key will be "forwarded" to the <term> rule. For example, if a '/' is typed when an <arex> is expected, it will be passed to <term>, where it will generate a Quo (quotient) node.

Note we have made some changes to abstract syntax: instead of one Appl node we now have different nodes for the different operations (Sum, Dif, etc.). This simplifies unparsing.

We will use a slightly different notation for transaction grammars that is more compact, since it merges the analysis and synthesis parts of the rule. For example, the rule for <arex> is:

```
<arex>
  + (Sum) -> Left:<arex> + Right:<term>
  - (Dif) -> Left:<arex> - Right:<term>
  else <factor>
```

This say that a '−' key creates a Sum node, whose Left component is an <arex> and whose Right component is a <term>. Table 3 shows most of the transaction grammar for the sample language; only routine parts have been omitted.

II. Requirements

What sort of tables are needed? We will consider the data structures needed to support language-independent (A) unparsing and (B) syntax-directed editing. This will allow us to define the necessary relations.

TABLE 3. Transaction Grammar for Sample Language

```
<program> =    <expr>

    <expr>:
               = (Eql) → Left:<arex> = Right:<arex>
               n (Neq) → Left:<arex> = Right:<arex>
                 :
               else <arex>

    <arex>:
               + (Sum) → Left:<arex> + Right:<term>
               − (Dif) → Left:<arex> − Right:<term>
               else <term>

    <term>:
               * (Prd) → Left:<term> × Right:<factor>
               / (Quo) → Left:<term> ÷ Right:<factor>
               else <factor>

    <factor>:
               # (Con) → LitVal:<chars>
               c (Call) → Rator:<chars> Rand:<factor>
               ( (Par) → ( Exp:<expr> )

               i (ConEx) → %I%N (if Cond:<expr>
                 %N then Conseq:<expr>
                 %N else Alt:<expr> ) %O

               l (Block) → %I let BndVar:<chars> = BndVal:<expr>
                 %N Body:<expr>  %O

               f (FunDef) → %I func FunName:<chars> FunFormal:<chars> = FunBody:<expr>
                 %N FunScope:<expr> | %O

               else (Var) -> Ident:<chars>
```

A. Unparsing

1. *Alternates*

For each syntactic category, e.g., <expr>, we must be able to select an image-template based on the kind of node (the node *variant*). E.g.

Block ⇒ **let** - = - -

ConEx ⇒ (**if** - **then** - **else** -)

:

These are called ***alternates*** of the syntactic category.

2. *Image-templates*

For each alternate, we need an *image-template,* which is a sequence of *items*. Items are either *terminals* or *nonterminals*.

a) *Terminals* are strings of characters and formatting commands (e.g., tabs and newlines). E.g.,

"%I%N(**if** "

b) *Nonterminals* must specify:

i. An associated ***selector relation,*** e.g., Cond, Conseq or Alt for ConEx nodes.

ii. A syntactic category determining what can legally occur in that position. e.g., <expr> for the Cond part of a ConEx. This is needed for:

- Prompting for the required category
- Determining the format by which to display a component

c) There are two kinds of image templates:

i. *Simplex*

ii. *Complex*

i. A ***simplex*** template is composed of a single nonterminal, possibly surrounded by terminal items. e.g.

"(" <expr> ")"

It does not break a node into its components for unparsing, but relies on the nonterminal's rule to do this.

ii. A *complex* template breaks a node into its components and unparses each component according to a given nonterminal. These unparsed results are catenated with appropriate terminals between and around them. e.g.

"%I%N (**if** " Cond:<expr>
"%N then" Conseq:<expr>
"%N **else** Alt:<expr> ") %O"

3. Display Control

We will also need a means for displaying the context of the current node, i.e., the surrounding tree structure. We want to display the context of the current node, but without zooming in or out too frequently (which would confuse the user). Furthermore, we want to suppress sufficient detail so that the displayed region of the tree will fit on the display screen. Therefore we define parameters for controlling the display as depicted in Figure 1.

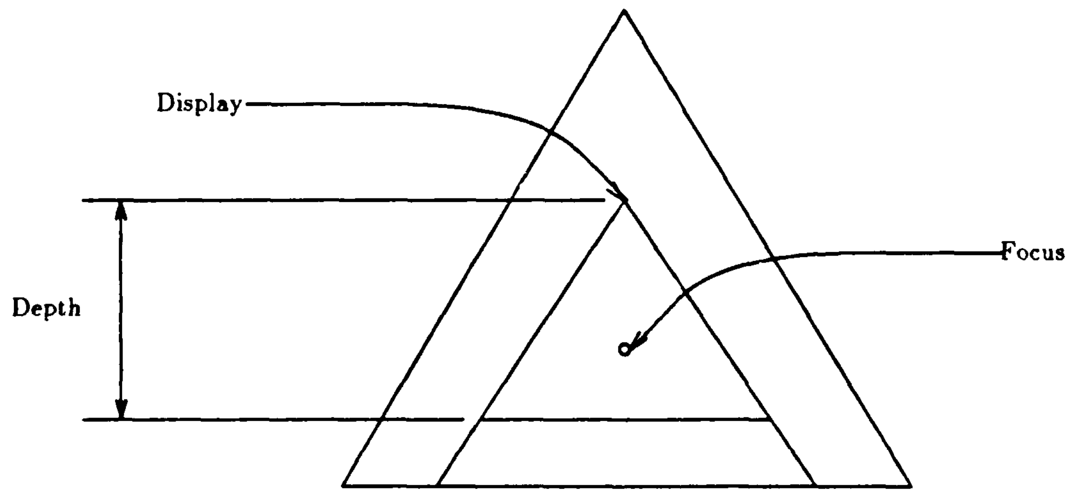


Figure 1. Parameters Controlling Display

'Display' represents the region of the tree presented on the screen. Detail below the 'Depth' level is suppressed so that the displayed region will fir on the screen. 'Focus' refers to the current node, to which commands refer. It is unparsed in a distinctive way (typically, in inverse video). The focus can be shifted anywhere within the displayed area. However, if the focus reaches the depth limit, then the display zooms in (i.e., the Display node shifts down). If the focus reaches the display root, then the display zooms out. This policy minimizes movement of Display.

B. Syntax-Directed Editing

We must consider:

1. Creation of program trees

2. Deletion of subtrees

3. Motion within the program tree

1. Creation

a. We need to be able to select a node kind out of those that are legal at a given position. E.g., if an <expr> is expected, then we must be able to select Var, Appl, ConEx, etc. Further, we must specify the command key associated with each selection.

b. We need to know the selector relations, so that undefined components can be created when a node is created. E.g., when a ConEx node is created, we need to link it to new undefined nodes by the Cond, Conseq and Alt relations.

c. We need to know which component (i.e., selector relation) is first (i.e., leftmost in display order) so that the focus can be shifted there. E.g., when a ConEx node is created, the focus should be shifted to its Cond component.

d. We need to know the next component (i.e., selector relation) in display order so that focus can be shifted to the right as each component is completed. E.g., when the Cond of a ConEx is finished, we shift to the Conseq, and when that is done to the Alt.

e. We need to be able to "chain" from grammar rule to grammar rule. E.g., Suppose '+' and '−' are legal command keys for <arex>, and '×' and '÷' are legal keys for <term>, and '(' and '#' are legal for <factor>. Also suppose a <factor> is a legal <term> and a <term> is a legal <arex>. Then, typing a '×' where an <arex> is required should chain to the <term> rule, and typing '#' where an <arex> is required should chain to the <factor> rule.

2. Deletion

a. We need to know the syntactic category expected after a deletion. For example, when a Quo node is deleted from the Conseq component of a ConEx, the user should be prompted for <expr>.

b. We need to know all selector relations for a node kind, so that all its components can be deleted.

3. Motion

a. We need to know next and previous nodes in display order. N.B., there is no next or previous inherent in the abstract structure; the display order is determined by the concrete syntax.

b. **We need to know first node in display order, for the in command.**

c. **We need to be able to get to the parent node, for the out command. Note that there is no one selector relation that links a node to its parent.**

III. Outline of Relations

Based on the preceding analysis we outline the relations required to support the necessary data structures. The requirements supported by a relation are listed in brackets following the description of the relation.

A. User Interface

The relations used for interfacing to the user are similar to those used in Parts I through V.

- Command (A, c)

  Agent A issues command c.

  Note that 'Command' has been changed from a one-place to a two-place relation; this simplifies command synchronization since it allows 'Command' to be called as a procedure, 'Command $\{c\}$'.

- Argument (s)

  s is the argument string.

- Error (A, s)

  Agent A reports error message s.

As usual we assume the existence of the 'display' procedure.

B. Kinds of Templates

Whenever we refer to a template T we mean the first item in the chain of items constituting the template.

- Template (T, V, N)

  T is the image template for node variant V under nonterminal N. [A1]

- DefaultTemplate (T, N)

  T is the default template for nonterminal N. [A1, B1e]

- Complex (T)

  T is a complex template. [A2c]

Thus, the data structures for a nonterminal such as <arex> look like this:

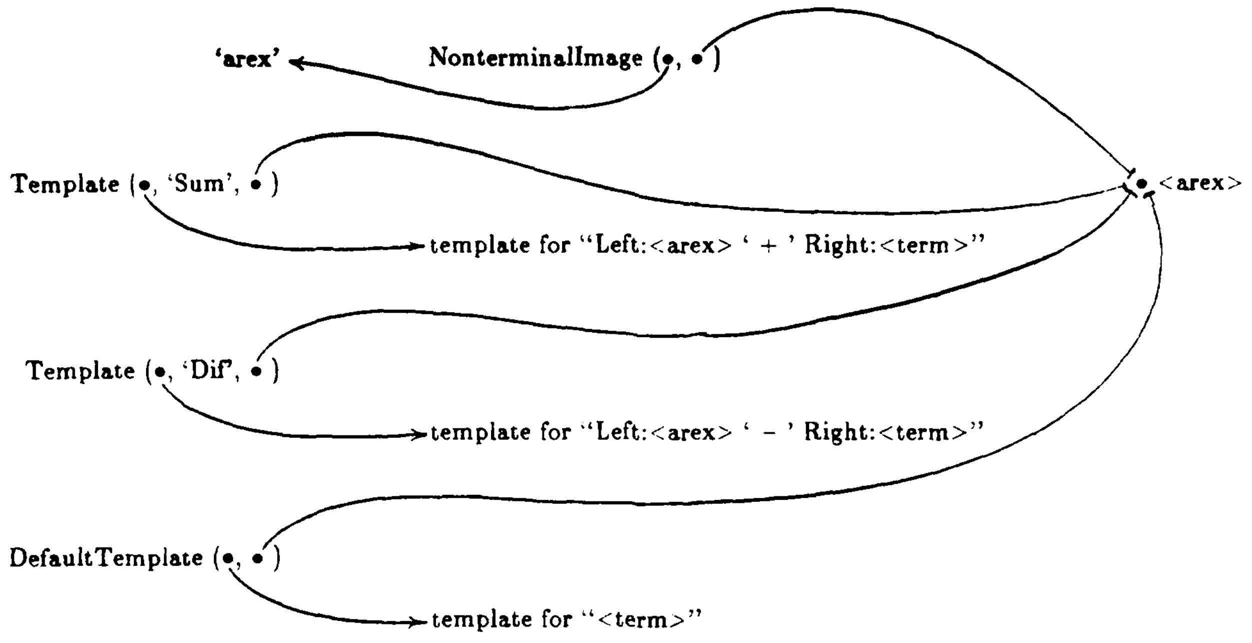


Figure 2. Data Structures Representing the Nonterminal <arex>

NonterminalImage is described in D, below.

C. Parts of Templates

These relations and those in the next section represent the parts of templates. In this section we describe the relations that represent the sequencing of items and the representation of terminal items.

- NextItem (*J*, *I*, *N*)

  J is the next item after *I* in an image template for nonterminal *N*. [A2]

- Terminal (*I*)

  I is a terminal item [A2a, A2b]

- TerminalImage (*S*, *I*)

  S is the image of terminal item I. [A2a]

D. Nonterminals

These relations represent nonterminals, including the node components to which they correspond, their syntactic category, and the string that represents them to the user.

- **Selector** (R, I, T)

  R is the selector for nonterminal item I in template T. [A2bi, B2a]

- **Category** (M, I, T)

  M is the syntactic category of nonterminal item I in template T. [A2bii]

- **NonterminalImage** (S, N)

  S is the image of nonterminal N. [A2bii]

The following figure displays the data structures corresponding to the template:

Left:<arex> ' − ' Right:<term>

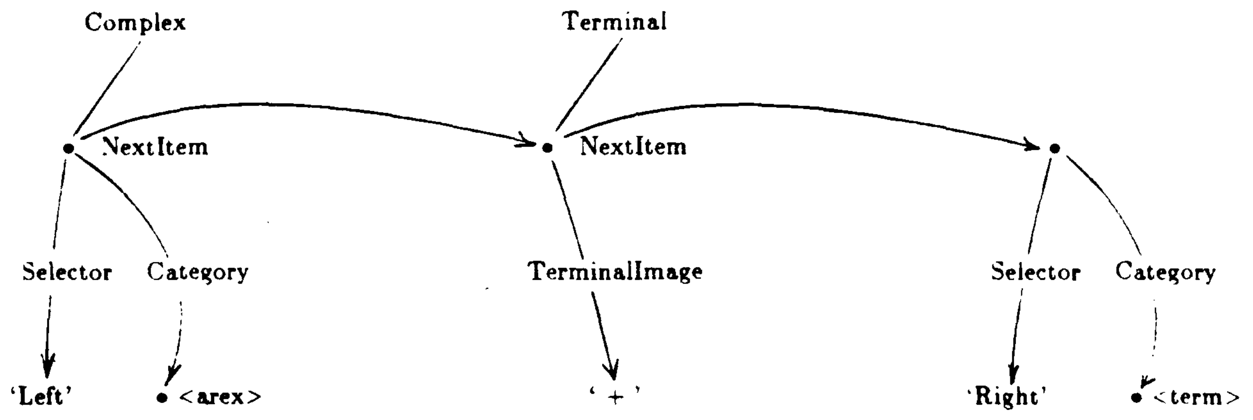


Figure 3. Data Structures for the Template: Left:<arex> ' − ' Right:<arex>

Note that for clarity we have omitted from the diagram the third fields of NextItem, Selector and Category.

E. Display Control

These relations control the region of the program displayed on the screen and control the suppression of low-level detail.

- DisplayRoot (E)

  E is the display root. [A3]

  This relation determines the region of the tree displayed on the screen. It is the node at which unparsing begins.

- **CurrentNode (E)**

  ***E* is the current node.** [A3]

  This relation determines the node that is the object of all editor commands (i.e., the *focus* of editing).

- DepthLimit (d)

  d is the depth limit. [A3]

  This parameter determines the depth beyond which detail is suppressed.

- CurrentDepth (l)

  l is the depth of current node. [A3]

  This is used to determine when the depth limit is exceeded.

- AssumedTemplate (T, V)

  T is assumed template for node type V. [A3, B2a]

  This is used to determine the template by which to unparse the display root.

These parameters are described in more detail later.

F. Variant Selection

The following relations are used for identifying language-dependent creation keys, forwarding them from rule to rule, and associating them with node variants.

- ExecutiveCommand (k)

  Key k is an executive command. [B1a]

  This is used for discriminating between the language-independent "executive command keys" and the language-dependent selector keys.

- VariantKey (k, V, N)

  Key k selects variant V of nonterminal N. [B1a]

  For example, '+' selects variant 'Sum' of <arex>, and '−' selects variant 'Dif' of <arex>.

- DefaultNT (N, M)

  Nonterminal N is the default alternative nonterminal M. [B1e]

  For example, <term> is the default nonterminal for <arex>.

G. Node Format

These relations specify the left-to-right (in display order) relationship among the selectors of a node. For example, in the template

'%I%N (**if** ' Cond:<expr>

'%N **then** ' Conseq:<expr>

'%N **else** ' Alt:<expr> ')%O'

the defined order of selectors is (Cond, Conseq, Alt). Note that we might easily have a different template that displays the components in a different order:

'%I%N' Conseq:<expr> ', **if** ' Cond:<expr>

'%N' Alt:<expr> ', **otherwise** '%O%N'

This template defines the order (Conseq, Cond, Alt). The relations describing selector order are:

- FirstSelector (*R*, *T*)

  R is first selector for template *T*. [B1b, B1c, B2b, B3b]

  For example, "Cond" is the first selector for the template for conditional expressions.

- NextSelector (*S*, *R*, *T*)

  S is next selector after *R* in template *T*. [B1b, B2b, B3a, B1c]

  For example, "Conseq" is the selector after "Cond" in the conditional expression template.

- LastSelector (*R*, *T*)

  R is the last selector for template *T*. [B1b, B2b, B3a, B1c]

  For example, "Alt" is the last selector in the conditional expression template.

H. Node Structure

The following relations define the actual format of nodes. This is a difference from previous reports in this series. Rather than using individual predicates to tag each node type, Con (*E*), Block (*E*), etc., we now use a single Type relation to attach the type. Hence we have Type ("Con", *E*), Type ("Block", *E*), etc. (the tag is simply a character string). Similarly, instead of representing node components by a number of specialized relations, LitVal (*V*, *E*), BndVar (*N*, *E*), etc., we now use a single Component relation, so we

have Component (V, "LitVal", E), Component (N, "BndVar", E), etc. Hence component relations are simply strings.

- Type (V, E)

  V is the type of node E. {A1, B1, B3b}

- Component (X, R, Y)

  X is the R component of Y. {B2a, B3c}

This completes the description of the principal relations used in the table-driven syntax-directed editor. viz., those relations that define the tables. There will of course be a number of *active* relations used to control the activities of the editor; these will be introduced as they are needed.

IV. Rules for the Table-Driven Syntax-Directed Editor

In this section we present most of the 53 rules required to implement the editor (the complete system is listed in Appendix A). Many of the rules are largely self explanatory, so they are presented with minimal commentary.

A. Focus Movement

These commands shift the focus under control of the template and node description. They all make use of the procedure 'SetCurrentNode', which shifts the focus and in addition determines if this focus movement requires the display root to be moved.

1. *in Command*

For an **in** command, we follow the branch for the first selector for the template associated with the focus's node type:

*Command (return, **in**), CurrentNode (E), Type (V, E), AssumedTemplate (T, V),
FirstSelector (R, T), Component (X, R, E)
$\Rightarrow$ {SetCurrentNode {X}; Command (return, **display**)}.

The **in** command, like all navigation commands, issues the **display** command, which refreshes the screen. We use ';' to indicate sequential action, since we want the focus to be shifted before the display is refreshed.

Also note that by passing 'return' to the Command procedure in the effect part of the rule we have made Command tail-recursive. The effect part could have been written recursively, but it would be less efficient:

$$\cdots \Rightarrow \{\text{SetCurrentNode } \{X\};\ \text{Command } \{\textbf{display}\};\ \text{return (Nil)}\}.$$

Error conditions can be handled by an **else** rule:

else *Command (return, **in**)

⇒ Error (return, "No inner component.").

We assume 'Error' is a procedure that takes appropriate error reporting action. In general we omit showing the error handling rules, since they are routine.

2. out Command

The **out** command is very simple; we move to the parent of the current focus:

*Command (return, **out**), CurrentNode (X), Component (X,˜–,˜E)

⇒ {SetCurrentNode {E}; Command (return, **display**)}.

In effect the rules says, "If the current node is some component of another node, then move to that other node."

3. next Command (next on same level)

For then **next** command we determine the selector corresponding to the current node and, via the assumed template for the parent node type, find the selector that follows the current one in that template. This allows selecting the new current node.

*Command (return, **next**), CurrentNode (X), Component (X, R, E), Type (V, E),

AssumedTemplate (T, V), NextSelector (S, R, T), Component (Y, S, E)

⇒ {SetCurrentNode {Y}; Command (return, **display**)}.

4. next Command (next on level above)

If there is *not* a next selector at the current level in the tree, then we shift the focus to the parent of the

current node and reissue the **next** command:

***Command (return, next)**, CurrentNode (X), Component (X, R, E), Type (V, E),

AssumedTemplate (T, V), ¬NextSelector (–, R, T)

⇒ {SetCurrentNode {E}; Command (return, **next**)}.

Thus, when we hit the **next** key, we will continue zooming out until either we reach a level where there is a next component, or we reach the root. The **prev** command is exactly analogous.

B. Focus Motion

1. Simple Version

In the simplest definition of the SetCurrentNode procedure simply moves the focus:

*SetCurrentNode (return, X), *CurrentNode (E)

⇒ CurrentNode (X), return (Nil).

The reason we have defined SetCurrentNode is to permit a more complicated version that shifts the DisplayRoot if necessary.

2. Display Parameters

To understand the operation of SetCurrentNode it is necessary to understand the role of the various display parameters (see Figure 4).

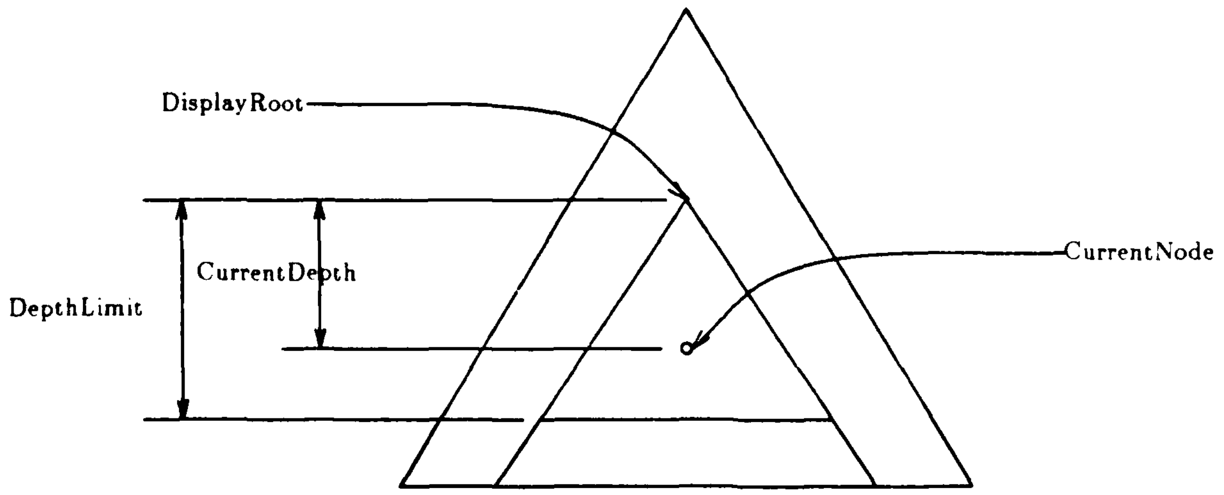


Figure 4. Display Parameters

The parameters are as described before (Figure 1) except that CurrentDepth records the current depth at which unparsing is taking place. When this exceeds DepthLimit, further unparsing is suppressed.

3. Upward Motion: Shift Needed

When an attempt is made to shift the focus to the parent of the current display root, then we are moving out of the displayed region. Therefore, the display root is moved to the parent of the new focus. The result is that the display gradually "zooms out" as the focus is shifted up the tree. This zooming may be too gradual when one is moving rapidly up the tree. A more complicated rule would locate the new root several above the new focus, so that it wouldn't have to shift so often.

> *SetCurrentNode (return, E), *CurrentNode (X), Component (X, –, E),
> *DisplayRoot (X), *CurrentDepth (–), Component (E, –, D)
>
> ⇒ DisplayRoot (D), CurrentNode (E), CurrentDepth (1), return (Nil).

4. Upward Motion: No Shift Needed

When we haven't reached the root, we simply set the new focus.

> **else** *SetCurrentNode (return, E), *CurrentNode (X), Component (X, –, E), *CurrentDepth (l)
>
> ⇒ CurrentNode (E), CurrentDepth ($l-1$), return (Nil).

5. Downward Motion: Shift Needed

When the focus has reached the depth limit, it is moving out of the bottom of the displayed region of the tree. Therefore it is necessary for the display to "zoom in" to keep the displayed region visible. This is accomplished by shifting the display root to the parent of the new focus. This rapid zooming works well when the user is moving down into the tree rapidly.

> *SetCurrentNode (return, X), *CurrentNode (E), Component (X, –, E), *DisplayRoot (–),
> *CurrentDepth (l), DepthLimit (d), **if** ($l \geq d$)
>
> ⇒ CurrentNode (X), DisplayRoot (E), CurrentDepth (1), return (Nil).

6. *Downward Motion: No Shift Needed*

When the depth limit has not been reached, the focus is simply shifted to the new node.

```
*SetCurrentNode (return, X), *CurrentNode (E), Component (X, –, E),
 DepthLimit (d), CurrentDepth (l), if (l< d)

⇒ CurrentNode (X), CurrentDepth (l+1), return (Nil).
```

7. *Arbitrary Motion*

All of the preceding cases assume that the focus is being shifted to a point neighboring the old focus. However, search commands may move the focus large distances. In these cases we set the display root two nodes above the new focus, provided the tree extends up that far, otherwise it is set as high as possible.

```
*SetCurrentNode (return, E), *CurrentNode (–), *CurrentDepth (–), *DisplayRoot (–)

⇒ {CurrentNode (E);
   Component (E, –, F), Component (F, –, G)  ! set root two above
     ⇒ CurrentDepth (2), DisplayRoot (G)
   else Component (E, –, F)  ! set root one above
     ⇒ CurrentDepth (1), DisplayRoot (F)
   else CurrentDepth (0), DisplayRoot (E);  ! set root here
   return (Nil) }
```

C. Editing

1. *Node Deletion*

We must "untag" the node (making it undefined), and sever all connections between it and its descendents.

```
*Command (return, delete), CurrentNode (E), *Type (V, E)
⇒ Purge (return, E).
```

The Purge relation removes all the relevant Component tuples:

```
*Purge (return, $E$), Component ($X$, –, $E$)
$\Rightarrow$ ¬Component ($X$, –, $E$), Purge (return, $E$)

else *Purge (return, $E$)
$\Rightarrow$ Command (return, display).
```

D. Program Entry

When a key is struck we must ensure it is not an executive command (**delete**, **in**, **out**, etc.). To do this we will use a relation ExecutiveCommand, which contains all the executive command characters. When a program entry key is struck, we begin searching for its meaning with the nonterminal expected at that point.

1. Initiate Search

We ensure that the key is not an executive command and that the current node is undefined. Then we determine from the template associated with the type of its parent the syntactic category expected at this point. The search begins with the selector keys associated with this category.

```
*Command (return, $k$), ¬ExecutiveCommand ($k$), CurrentNode ($E$),
 ¬Type (–, $E$), Component ($E$, $R$, $P$), Type ($V$, $P$),
AssumedTemplate ($T$, $V$), Selector ($R$, $I$, $T$), Category ($M$, $I$, $T$)

 $\Rightarrow$ {OriginalNode ($E$);
   ProcessKey {$k$, $E$, $M$};
    ¬OriginalNode ($E$);
   return (Nil) }.
```

The relation OriginalNode saves the original node for restoration if the entered key is illegal. This is necessary since the forwarding process (optimistically) creates and focuses on subnodes as it forwards the key from nonterminal to nonterminal. See the rule for illegal key processing (below) for the use of OriginalNode.

2. K , Search

If the given key is one of the selectors for this nonterminal, then the appropriate node type is instantiated.

```
*ProcessKey (return, k, E, M), VariantKey (k, V, M), Template (T, V, M)
⇒ {Type (V, E); Instantiate (return, T, E)}
```

If it is not, then it is forwarded to another nonterminal, if one is specified, otherwise an error occurs. There are two cases depending on whether or not the template is complex:

```
else *ProcessKey (return, k, E, M), DefaultNT (N, M), ¬Complex (T)
⇒ ProcessKey (return, k, E, N)

else *ProcessKey (return, k, E, M), DefaultVar (V, M), DefaultTemplate (T, M),
  Complex (T), FirstSelector (R, T), Selector (R, I, T), Category (N, I, T)
⇒ {Type (T, V);
    Instantiate {T, E};
    Component (X, R, E) ⇒ ProcessKey (return, k, X, N)
```

Note that the preceding rule automatically instantiates nodes. It is this action that necessitates the use of OriginalNode in the following rule; whatever nodes have been added are deleted:

```
else *ProcessKey (return, k, E, M), OriginalNode (X), Type (–, X)
⇒ {SetCurrentNode {X};
    Command {delete};
    Error (return, "Illegal Key: " ^ k) }.
```

The presence of instantiated nodes are detected by a defined Type. The following rule handles the case where no nodes were instantiated.

```
else *ProcessKey (return, k, E, M)  ! nodes have not been instantiated
⇒ Error (return, "Illegal Key: " ^ k).
```

3. Node Instantiation: First Selector

The first component of the new node is distinguished from the rest because the focus is automatically

shifted to this component. This facilitates left-to-right program entry. The following rule allocates a new node, makes it the first component, and makes it the new focus:

*Instantiate (return, T, E), FirstSelector (R, T), *Avail (X)

$\Rightarrow$ Component (X, R, E), SetCurrentNode $\{X\}$, InstantiateRest (return, R, T, E).

4. *Node Instantiation: Other Selectors*

The remaining component nodes are created by an iterative process. When this has completed, the altered tree is unparsed, so that the user can see the template for the new node.

*InstantiateRest (return, R, T, E), NextSelector (S, R, T), *Avail (X)

$\Rightarrow$ Component (X, S, E), InstantiateRest (return, S, T, E).

*InstantiateRest (return, R, T, E), ¬NextSelector (–, R, T)

$\Rightarrow$ Command (return, **display**).

E. String Entry

The command processor recognizes the spacebar command and accumulates into the Argument relation all characters typed up to the next return character. This command is allowed only if <chars> is the non-terminal associated with the current node.

For example, consider the grammar rule:

→ <chars>

$\Rightarrow$ Con (LitVal=<chars>)

The generated tuples are:

Type (Con, E),

Component (V, LitVal, E)

Type (Chars, V),

Component ("$c_1c_2 \cdots c_n$", "charval", V)

The extra level of indirection is necessary to allow editing the string value.

1. Program Entry: Spacebar Command

If the current node is undefined, but a string is expected, then the accumulated argument is made the string associated with the node, which is now of type Chars.

> *Command (return, **spacebar**), *Argument (s), CurrentNode (E),
>
> ¬Type (–, E), Component (E, R, P), Type (V, P),
>
> AssumedTemplate (T, V), Selector (R, I, T), Category (String, I, T)
>
> ⇒ Type (Chars, E), Component (s, "charval", E), Command (return, **display**).

The rule arrangement in Appendix A is slightly different, but not significantly so.

2. Alternate String Entry

The need to type the spacebar could be eliminated by making all commands (both executive and selector) nonprintable characters (e.g., control codes). Then, any printable character would automatically start string accumulation. Alternately, positioning the cursor on an undefined node of type <chars> could automatically trigger string accumulation.

F. Unparsing

The following sections describe the rules for unparsing the visible region of the program tree. Unparsing is initiated by the **display** command, which may be issued by the user or by the system (e.g., during editing).

1. display Command

Begin at the display root:

> *Command (return, **display**), DisplayRoot (E), Type (–, E)
>
> ⇒ Unparse (E), WaitImage (return, E).

The Unparse relation requests the unparsing to start at E. The unparsed form is returned in FinalImage:

> ***WaitImage (return, E), *FinalImage (s, E)**
>
> **⇒ return (Display {s}).**

2. Begin Unparsing

Select a template based on the type of node:

*Unparse (E), Type (V, E), AssumedTemplate (T, V)

$\Rightarrow$ Scan (E, T, 0).

This requests E to be unparsed (scanned) according to template T. The '0' indicates that the current display depth is zero.

3. Begin Image Accumulation

To scan according to a template, we begin processing the first item in the template, starting with an empty image:

*Scan (E, T, l)

$\Rightarrow$ ProcessItem (E, T, "", l).

4. Terminals

A terminal string is added to the image accumulated so far:

*ProcessItem (E, I, s, l), Terminal (I), TerminalImage (t, I)

$\Rightarrow$ MoveRight (E, I, s ^ t, l).

5. Move Right

The MoveRight relation finds the next item in the template, or, if there is none, completes processing of the image.

*MoveRight (E, I, s, l), NextItem (J, I, N),

$\Rightarrow$ ProcessItem (E, J, s, l)

else *MoveRight (E, I, s, l)

$\Rightarrow$ Image (s, E).

6. Nonterminal: Component Node

Find the associated selector and through it initiate unparsing of the selected component.

*ProcessItem (E, I, s, l), ¬Terminal (I), Selector (R, I, T),

Component (X, R, E), Category (N, I, T), Complex (T)

⇒ Pending (E, I, s, X, l), FindTemplate (X, N, $l+1$).

The Pending relation waits until the component is unparsed. FindTemplate requests unparse of the component at display level $l+1$.

7. Nonterminal: Entire Node

This occurs in simplex (as opposed to complex) templates. It is triggered by the absence of a selector for the nonterminal item.

*ProcessItem (E, I, s, l), ¬Terminal (I), Category (N, I, T), ¬Complex (T)

⇒ Pending (E, I, s, E, l), FindTemplate (E, N, l).

8. Nonterminal: Complete Processing

When a final image has been computed for a pending nonterminal, we move right to the next item in the template:

*Pending (E, I, s, X, l), *FinalImage (t, X)

⇒ MoveRight (E, I, s ^ t, l).

9. Find Template: Undefined Node

If the node is undefined, its image is the name of the nonterminal expected:

*FindTemplate (E, N, l), ¬Type (–, E), NonterminalImage (s, N)

⇒ Image ("<" ^ s ^ ">", E).

The Image relation is an intermediary before FinalImage, which is discussed later.

10. Find Template: Suppress Detail

If the subtree is below the depth limit, then replace it by an ellipsis:

*FindTemplate (E, N, l), Type (–, E), DepthLimit (d), if $l>d$

$\Rightarrow$ Image ("...", E).

Note that this rule is not used on undefined nodes.

11. Find Template: Selected Template

Select a template based on the type of the node:

*FindTemplate (E, N, l), Type (V, E), Template (T, V, N), DepthLimit (d), if $l\leq d$

$\Rightarrow$ Scan (E, T, l).

12. Find Template: Default Template

If there is no template corresponding to the node type, then use the default template:

*FindTemplate (E, N, l), Type (V, E), ¬Template (–, V, N),

DefaultTemplate (T, N), DepthLimit (d), if $l\leq d$

$\Rightarrow$ Scan (E, T, l).

13. Find Template: Leaves

We must handle specially the "leaves," that is, the nodes corresponding to the <chars> nonterminal:

*FindTemplate (E, String, l), Type (Chars, E), Component (s, "charval", E)

$\Rightarrow$ Image (s, E).

14. Final Image from Image

The final image is computed from the image by toggling inverse video when we pass the focus node:

```
*Image ($s$, $E$), CurrentNode ($E$), if ($s \neq$ Nil $\wedge$ first $s \neq$ invert_on)
    ⇒ FinalImage (invert_on ^ $s$ ^ invert_off, $E$)

else *Image ($s$, $E$)
    ⇒ FinalImage ($s$, $E$).
```

V. Conclusions

The investigation reported in "Experience with Ω: Implementation of a Prototype Programming Environment," Parts I through VI, has given us considerable experience in the use of Ω for a nontrivial application. We have gained some insight into the difficulties of programming a complex, highly-concurrent system in an object-oriented language. This has convinced us that object-oriented languages need some powerful linguistic mechanisms to help programmers think about time and control the relationship of events in time.

We now believe that some form of modal or temporal logic, *built into the language*, is the best approach to this problem. In particular, in future research we intend to design linguistic structures that define *temporal domains* in much the same way that the control structures and scope structures of conventional languages define control-flow and naming domains. The goal of the linguistic structures is to avoid the complicated reasoning common to the use of first-order temporal logics.

VI. References

[MacLennan83] MacLennan, B. J., A View of Object-Oriented Programming, Naval Postgraduate School Computer Science Department Technical Report NPS52-83-001, February 1983.

[MacLennan84] MacLennan, B. J., The Four Forms of Ω: Alternate Syntactic Forms for an Object-Oriented Language, Naval Postgraduate School Computer Science Department Technical Report NPS52-84-026, December 1984.

[MacLennan85a] MacLennan, B. J., A Simple Software Environment Based on Objects and Relations, *Proc. of ACM SIGPLAN 85 Conf. on Language Issues in Prog. Environments*, June 25-28, 1985, and Naval Postgraduate School Computer Science Department Technical Report NPS52-85-005, April 1985.

[MacLennan85b] MacLennan, B. J., Experience with Ω: Implementation of a Prototype Programming Environment Part I, Naval Postgraduate School Computer Science Department Technical Report NPS52-85-006, May 1985.

[MacLennan85c] MacLennan, B. J., Experience with Ω: Implementation of a Prototype Programming Environment Part II, Naval Postgraduate School Computer Science Department Technical Report NPS52-85-015, December 1985.

[MacLennan86a] MacLennan, B. J., Experience with Ω: Implementation of a Prototype Programming Environment Part III, Naval Postgraduate School Computer Science Department Technical Report NPS52-86-004, January 1986.

[MacLennan86b] MacLennan, B. J., Experience with Ω: Implementation of a Prototype Programming Environment Part IV, Naval Postgraduate School Computer Science Department Technical Report NPS52-86-007, January 1986.

[MacLennan86c] MacLennan, B. J., Experience with Ω: Implementation of a Prototype Programming Environment Part V, Naval Postgraduate School Computer Science Department Technical Report NPS52-86-009, January 1986.

[McArthur84] McArthur, Heinz M., *Design and Implementation of an Object-Oriented, Production-Rule Interpreter*, MS Thesis, Naval Postgraduate School Computer Science Department, December 1984.

[Ufford85] Ufford, Robert P., *The Design and Analysis of a Stylized Natural Grammar for an Object Oriented Language (Omega),* MS Thesis, Naval Postgraduate School Computer Science Department, June 1985.

APPENDIX A: Universal Syntax-Directed Editor

The following is a loadable input file ('PI6.USDE') for the prototype universal syntax-directed editor described in this report. It is accepted by the McArthur interpreter [McArthur84], which differs in a few details from the Ω notation used in this report (see [MacLennan84]).

```
!
! USDE -- Universal Syntax Directed Editor
!

! RELATION DECLARATIONS

! User Interface

newrelation {"Command"};
newrelation {"Argument"};
newrelation {"Error"};

! Kinds of Templates

newrelation {"Template"};
newrelation {"DefaultTemplate"};
newrelation {"Complex"};

! Parts of Templates

newrelation {"NextItem"};
newrelation {"Terminal"};
newrelation {"TerminalImage"};

! Nonterminals

newrelation {"Selector"};
newrelation {"Category"};
newrelation {"NonterminalImage"};

! Display Control

newrelation {"Root"};
newrelation {"DisplayRoot"};
newrelation {"CurrentNode"};
newrelation {"OriginalNode"};
newrelation {"DepthLimit"};
newrelation {"CurrentDepth"};
newrelation {"AssumedTemplate"};

! Variant Selection

newrelation {"ExecutiveCommand"};
newrelation {"VariantKey"};
newrelation {"DefaultNT"};
newrelation {"DefaultVar"};

! Node Format

newrelation {"FirstSelector"};
newrelation {"NextSelector"};
newrelation {"LastSelector"};

! Node Structure

newrelation {"Type"};
newrelation {"Component"};

! Activities

newrelation {"SetCurrentNode"};
newrelation {"Purge"};
newrelation {"Instantiate"};

newrelation {"Image"};

newrelation {"FinalImage"};

newrelation {"FindTemplate"};

newrelation {"Pending"};

newrelation {"MoveRight"};

newrelation {"ProcessItem"};

newrelation {"Scan"};

newrelation {"Unparse"};

newrelation {"WaitImage"};

newrelation {"InstantiateRest"};

newrelation {"ProcessKey"};

newrelation {"CreateFirst"};

newrelation {"CreateRest"};

newrelation {"CreateRoot"}.

! Format Control Strings

define {root. "NL". "
"};

define {root. "invert_on". "{"};

define {root. "invert_off". "}"}.

! chars -- a special builtin nonterminal

define {root. "chars". newobj {}}.

AssumedTemplate (newobj {}. "chars");

NonterminalImage ("chars". chars).

! Table of Executive Commands

ExecutiveCommand ("in");
ExecutiveCommand ("out");

ExecutiveCommand ("next");

ExecutiveCommand ("prev");

ExecutiveCommand ("root");

ExecutiveCommand ("display");

ExecutiveCommand ("delete");

ExecutiveCommand ("begin").
```

```
! THE RULES

define {root. "UnivSDErules". <<

! Error Handler

if *Error (return. message)

-> return (displayn {message});

! Focus Movement:

! in Command

if *Command (return. "in"). CurrentNode (E). Type (V. E). AssumedTemplate (T. V).

 FirstSelector (R. T). Component (X. R. E)

-> {SetCurrentNode {X}; Command (return. "display")}

else if *Command (return. "in")

-> Error (return. "No inner component");

! out Command

if *Command (return. "out"). CurrentNode (X). Component (X. -. E)

-> {SetCurrentNode {E}; Command (return. "display")}

else if *Command (return. "out")

-> Error (return. "No outer component");

! next Command  (next on same level)

if *Command (return. "next"). CurrentNode (X). Component (X. R. E). Type (V. E).

 AssumedTemplate (T. V). NextSelector (S. R. T). Component (Y. S. E)

-> {SetCurrentNode {Y}; Command (return. "display")};

! next Command  (next on level above)
```

```
if *Command (return. "next"). CurrentNode (X). Component (X. R. E). Type (V. E).
 AssumedTemplate (T. V). ~NextSelector (–. R. T)
-> {SetCurrentNode {E}: Command (return. "next")};

! next Command (no next node)

if *Command (return. "next"). CurrentNode (X). ~Component (X. R. E)
-> Error {"No next component"};

! prev Command (prev on same level)

if *Command (return. "prev"). CurrentNode (X). Component (X. R. E). Type (V. E).
 AssumedTemplate (T. V). NextSelector (R. S. T). Component (Y. S. E)
-> {SetCurrentNode {Y}: Command (return. "display")};

! prev Command (prev on level above)

if *Command (return. "prev"). CurrentNode (X). Component (X. R. E). Type (V. E).
 AssumedTemplate (T. V). ~NextSelector (R. S. T)
-> {SetCurrentNode {E}: Command (return. "prev")};

! prev Command (no prev node)

if *Command (return. "prev"). CurrentNode (X). ~Component (X. R. E)
-> Error {"No prev component"};

! root Command

if *Command (return. "root"). Root (E)
-> {SetCurrentNode {E}: Command (return. "display")};

! Upward Motion: Shift Needed

if *SetCurrentNode (return. E). *CurrentNode (X). Component (X. –. E). *DisplayRoot (X).
 *CurrentDepth (–). Component (E. –. D)
```

```
-> DisplayRoot (D). CurrentNode (E). CurrentDepth (1). return (Nil)

! Upward Motion: No Shift Needed

else if *SetCurrentNode (return. E). *CurrentNode (X). Component (X. -. E). *CurrentDepth (I)
-> CurrentNode (E). CurrentDepth (I-1). return (Nil)

! Downward Motion: Shift Needed

else if *SetCurrentNode (return. X). *CurrentNode (E). Component (X. -. E).
 *DisplayRoot (-). *CurrentDepth (I). DepthLimit (d). I >= d
-> CurrentNode (X). DisplayRoot (E). CurrentDepth (1). return (Nil)

! Downward Motion: No Shift Needed

else if *SetCurrentNode (return. X). *CurrentNode (E). Component (X. -. E).
 DepthLimit (d). *CurrentDepth (I). I < d
-> CurrentNode (X). CurrentDepth (I+1). return (Nil)

! Arbitrary Motion: Default Case

else if *SetCurrentNode (return. E). *CurrentNode (-). *CurrentDepth (-). *DisplayRoot (-)
-> {CurrentNode (E);
   if Component (E. -. F). Component (F. -. G)  ! set root two above
     -> CurrentDepth (2). DisplayRoot (G)
   else if Component (E. -. F)  ! set root one above
     -> CurrentDepth (1). DisplayRoot (F)
   else CurrentDepth (0). DisplayRoot (E);  ! set root here
   return (Nil)
   };

! Editing Node Deletion

if *Command (return. "delete"). CurrentNode (E). *Type (V. E)
```

```
-> Purge (return, E);

if *Purge (return, E), Component (X, -, E)
-> ~Component (X, -, E), Purge (return, E)

else if *Purge (return, E)
-> Command (return, "display");

! Program Entry: Initiate Search

if *Command (return, k), ~ExecutiveCommand (k), CurrentNode (E), ~Type (-, E),
 Component (E, R, P), Type (V, P), AssumedTemplate (T, V), Selector (R, I, T),
 Category (M, I, T)
-> {OriginalNode (E);
   ProcessKey {k, E, M};
   ~OriginalNode (E);
   return (Nil)};

! Program Entry: Key Search

if *ProcessKey (return, k, E, M), VariantKey (k, V, M), Template (T, V, M)
-> {Type (V, E); Instantiate (return, T, E) }

else if *ProcessKey (return, k, E, M), DefaultNT (N, M), DefaultTemplate (T, M), ~Complex (T)
-> ProcessKey (return, k, E, N)

else if *ProcessKey (return, k, E, M), DefaultVar (V, M), DefaultTemplate (T, M),
 Complex (T), FirstSelector (R, T), Selector (R, I, T), Category (N, I, T)
-> {Type (V, E);
   Instantiate {T, E};
   if Component (X, R, E) -> ProcessKey (return, k, X, N)
  }
```

```
! Process Character Entry

else if *ProcessKey (return. " ". E. chars). *Argument (s)

-> Type ("chars". E). Component (s. "charval". E). Command (return. "display")

else if *ProcessKey (return. " ". E. M). *Argument (-)

-> Error (return. "Characters not expected")

! Process Illegal Key

else if *ProcessKey (return. k. E. M). OriginalNode (X). Type (-. X)

-> {SetCurrentNode {X};

   Command {"delete"};

   Error (return. "Illegal Key: " + k)

  }

else if *ProcessKey (return. k. E. M)  ! nodes have not been instantiated

-> Error (return. "Illegal Key: " + k);

! Node Instantiation First Selector

if *Instantiate (return. T. E). FirstSelector (R. T)

-> CreateFirst (return. T. E. R. newobj {});

if *CreateFirst (return. T. E. R. X)

-> Component (X. R. E). SetCurrentNode {X}. InstantiateRest (return. R. T. E);

! Node Instantiation: Other Selectors

if *InstantiateRest (return. R. T. E). NextSelector (S. R. T)

-> CreateRest (return. R. T. E. S. newobj {});

if *CreateRest (return. R. T. E. S. X)

-> Component (X. S. E). InstantiateRest (return. S. T. E);
```

```
if *InstantiateRest (return, R, T, E), ~NextSelector (-, R, T)
-> Command (return, "display");

! Alternate String Entry

! Unparsing: display Command

if *Command (return, "display"), DisplayRoot (E), Type (-, E)
-> Unparse (E), WaitImage (return, E)

else if Command (return, "display"), DisplayRoot (X), Component (X, -, E), *CurrentDepth (-)
-> DisplayRoot (E), CurrentDepth (1);

if *WaitImage (return, E), *FinalImage (s, E)
-> return (displayn {s});

! Begin Unparsing
! Select a template based on the type of node:

if *Unparse (E), Type (V, E), AssumedTemplate (T, V)
-> Scan (E, T, 0);

! Begin Image Accumulation

if *Scan (E, T, I)
-> ProcessItem (E, T, "", I);

! Terminals

if *ProcessItem (E, I, s, I), Terminal (I), TerminalImage (t, I)
-> MoveRight (E, I, s + t, I);

! Move Right

if *MoveRight (E, I, s, I), NextItem (J, I, N)
```

```
-> ProcessItem (E, J, s, l)

else if *MoveRight (E, I, s, l)

-> Image (s, E);

! Nonterminal: Component Node

if *ProcessItem (E, I, s, l), ~Terminal (I), Selector (R, I, T),

 Component (X, R, E), Category (N, I, T), Complex (T)

-> Pending (E, I, s, X, l), FindTemplate (X, N, l+1);

! Nonterminal Entire Node

if *ProcessItem (E, I, s, l), ~Terminal (I), Category (N, I, T), ~Complex (T)

-> Pending (E, I, s, E, l), FindTemplate (E, N, l);

! Nonterminal: Complete Processing

if *Pending (E, I, s, X, l), *FinalImage (t, X)

-> MoveRight (E, I, s + t, l);

! Find Template: Undefined Node

if *FindTemplate (E, N, l), ~Type (-, E), NonterminalImage (s, N)

-> Image ("<" + s + ">", E);

! Find Template: Suppress Detail

if *FindTemplate (E, N, l), Type (-, E), DepthLimit (d), l>d

-> Image ("...", E);

! Find Template: Selected Template

if *FindTemplate (E, N, l), Type (V, E), Template (T, V, N), DepthLimit (d), l <= d

-> Scan (E, T, l);
```

```
! Find Template: Default Template

if *FindTemplate (E. N. I). Type (V. E). ~Template (-. V. N).
 DefaultTemplate (T. N). DepthLimit (d). I <= d
-> Scan (E. T. I);

! Find Template: Leaves

if *FindTemplate (E. chars. I). Type ("chars". E). Component (s. "charval". E)
-> Image (s. E);

! Final Image from Image

if *Image (s. E). CurrentNode (E). s != Nil. first [s] != invert_on
-> FinalImage (invert_on + s + invert_off. E)

else if *Image (s. E)
-> FinalImage (s. E);

! begin Command

if *Command (return. "begin")
-> CreateRoot (return. newobj {}. newobj {});

if *CreateRoot (return. E. P)
-> {Type ("prog". P). Root (P). Component (E. "*". P). DisplayRoot (P). CurrentNode (E);
   Command (return. "display")};

>>}.

act {UnivSDErules}.

! Initialize Parameters

DepthLimit (5);
```

```
CurrentDepth (1).

displayn {"Universal SDE loaded."}.
```

APPENDIX B: Table Builder for Universal SDE

The following is a loadable input file ('PI6.util') defining the table building and other utility rules for the universal syntax-directed editor. It assumes the definitions of Appendix A.

```
!
! Utility Relations and Rules for USDE
!

newrelation {"newobject"};
newrelation {"defnt"};
newrelation {"defnt2"};

newrelation {"CrTempl"};
newrelation {"CrTempl0"};
newrelation {"CrTempl1"};
newrelation {"CrTempl2"};
newrelation {"CrTempl3"};
newrelation {"CrTempl4"};

newrelation {"CrVar"};
newrelation {"CrDef"};
newrelation {"CrSimDef"};
newrelation {"DefSel"};

newrelation {"PrNT"};
newrelation {"PrVariant"};
newrelation {"PrSimDef"};
newrelation {"PrDef"};
newrelation {"PrTempl"};
newrelation {"PrTempl1"};
newrelation {"Script"};

newrelation {"PendScript"}.

define {root. "Pl6util". <<

! newobject {n} - defines a new object named 'n'

if *newobject (return. N)

-> {define {root. N. newobj {}}; return (Nil)};

! defnt {nt} - define nonterminal named 'nt'

if *defnt (return. S)

-> defnt2 (return. S. newobj{});

if *defnt2 (return. S. N)

-> {define {root. S. N}; NonterminalImage (S. N)};

! CrTempl {NT. templ-descr} - construct template; return 1st item

if *CrTempl (return. N. Nil)

-> {display {"Error: empty template for "}; displayn {N}}

else if *CrTempl (return. N. Z)

-> CrTempl0 (return. N. Z. newobj {});

if *CrTempl0 (return. N. Z. T)

-> CrTempl1 (return. N. Z. T. T);

if *CrTempl1 (return. N. Z. T. I)

-> {if IsList [first [Z]]

      -> CrTempl2 (return. N. first [rest [first [Z]]]. first [first [Z]].

           rest [Z]. T. I)

    else CrTempl3 (return. N. first [Z]. rest [Z]. T. I) };
```

```
if *CrTempl2 (return, N, M, R, Z, T, I)
-> {Selector (R, I, T), Category (M, I, T);
   DefSel {R, T};
   CrTempl4 (return, N, Z, T, I, N)
  };

if *CrTempl3 (return, N, S, Z, T, I)
-> Terminal (I), TerminalImage (S, I), CrTempl4 (return, N, Z, T, I, N);

if *CrTempl4 (return, N, Nil, T, I, N)
-> return (I)

else if *CrTempl4 (return, N, Z, T, I, N)
-> NextItem (CrTempl1 {N, Z, T, newobj {}}, I, N), return (I);

! CrVar {NT, key, type, template}
! Create variant

if *CrVar (return, N, K, V, Z)
-> {VariantKey (K, V, N), Template (CrTempl {N, Z}, V, N);
   if Template (T, V, N) -> AssumedTemplate (T, V), Complex (T);
   return (Nil)
  };

! CrDef {NT, default variant, template}
! Create default

if *CrDef (return, M, V, Z)
-> {DefaultVar (V, M), DefaultTemplate (CrTempl {M, Z}, M);
   if DefaultTemplate (T, M) -> AssumedTemplate (T, V), Complex (T);
   return (Nil)
  };
```

```
! CrSimDef {NT. left. NT'. right} - Create simplex default

if *CrSimDef (return. M. s. N. t)
-> {DefaultNT (N. M). DefaultTemplate (CrTempl {M. [s. ["*". N]. t]}. M);
   return (Nil)
  };

! DefSel {R. template} - Define next selector for template

if *DefSel (return. R. T). ~LastSelector (-. T)
-> FirstSelector (R. T). LastSelector (R. T). return (Nil);

if *DefSel (return. S. T). *LastSelector (R. T)
-> NextSelector (S. R. T). LastSelector (S. T). return (Nil);

! PrNT {NT} - print nonterminal's name

if *PrNT (return. N). NonterminalImage (S. N)
-> {displayn {""}; displayn {S + ":"}};

! PrVariant {NT. key} - print selected variant

if *PrVariant (return. N. K). VariantKey (K. V. N). Template(T. V. N). Complex (T)
-> {display {"'" + K + "' (" + V + ") -> "};
   return (PrTempl {T})
  };

! PrSimDef {NT} - print simplex default variant

if *PrSimDef (return. M). DefaultNT (N. M). DefaultTemplate (T. M)
-> {display {"else ( "}; display {N}; display {") -> "};
   return (PrTempl {T})
  };
```

```
! PrDef {NT} - print complex default variant

if *PrDef (return, M), DefaultVar (V, M), DefaultTemplate (T, M)
-> {display {"else (" + V + ") -> "};
   return (PrTempl {T})
   };

! PrTempl {item} - print template

if *PrTempl (return, I), Terminal (I), TerminalImage (S, I)
-> {display {" '" + S + "'"}; return (PrTempl1 {I})};

if *PrTempl (return, I), Selector (R, I, T), Category (M, I, T)
-> {display {" " + R + ":<"}; display{M}; display {">"};
   return (PrTempl1 {I})
   }.

if *PrTempl1 (return, I), ~NextItem (J, I, N)
-> displayn {" "}

else if *PrTempl1 (return, I), NextItem (J, I, N)
-> PrTempl (return, J);

! Script {s} - Execute Command Script s

if *Script (return, Nil)
-> return ("Done")

else if *Script (return, s), first [s] = " "
-> {displayn {"______________________________________________________"};
   displayn {" _" + first [rest [s]]};
   Argument (first [rest [s]]);
   Command {first [s]};
```

```
        if ~Command (-, -) -> Script (return, rest [rest [s]])
        else PendScript (return, rest [rest [s]])
        }

    else if *Script (return, s)
    -> {displayn {"________________________________________________________"};
        displayn {" " + first [s]};
        Command {first [s]};
        if ~Command (-, -) -> Script (return, rest [s])
        else PendScript (return, rest [s])
        };

    if *PendScript (return, s), ~Command (-, -), ~WaitImage (-, -)
    -> Script (return, s);

      }.

    act {PI6util}.
    displayn {"PI-6 Utilities activated."}.
```

APPENDIX C: Sample Transaction Grammar

The following is a loadable input file ('PI6.testgram') that constructs the grammar tables for the sample language used in the body of the report.

```
!
! Transaction Grammar for Simple Applicative Language
!

! Define Nonterminals

defnt {"program"};
defnt {"expr"};
defnt {"arex"};
defnt {"term"};
defnt {"factor"};
defnt {"var"}.

! <program> = '(prog) *:<expr>.

CrDef {program, "prog", [["*", expr]]}.

PrNT {program};
PrDef {program}.

! <expr>:
!   = (eql)  -> left:<arex> = right:<arex>
!   n (neq)  -> left:<arex> <> right:<arex>
!   else <arex>

CrVar {expr, "=", "eql", [["left", arex], " = ", ["right", arex]]};
CrVar {expr, "n", "neq", [["left", arex], " <> ", ["right", arex]]};
CrSimDef {expr, "", arex, ""}.
```

```
PrNT {expr};

PrVariant {expr, "="};

PrVariant {expr, "n"};

PrSimDef {expr}.


! <arex>:

!   + (sum)  -> left:<arex> + right:<term>

!   - (dif)  -> left:<arex> - right:<term>

!   else <term>


CrVar {arex, "+", "sum", [["left", arex], " + ", ["right", term]]};

CrVar {arex, "-", "dif", [["left", arex], " - ", ["right", term]]};

CrSimDef {arex, "", term, ""}.


PrNT {arex};

PrVariant {arex, "+"};

PrVariant {arex, "-"};

PrSimDef {arex}.


! <term>:

!   * (prd)  -> left:<term> x right:<factor>

!   / (quo)  -> left:<term> / right:<factor>

!   else <factor>


CrVar {term, "*", "prd", [["left", term], " x ", ["right", factor]]};

CrVar {term, "/", "quo", [["left", term], " / ", ["right", factor]]};

CrSimDef {term, "", factor, ""}.


PrNT {term};

PrVariant {term, "*"};

PrVariant {term, "/"};

PrSimDef {term}.
```

```
! <factor>:
! # (con) -> litval:<chars>
! c (call) -> rator:<chars> rand:<factor>
! ( (par) -> ( exp:<expr> )
! i (conex) ->
!     (if cond:<expr>
!      then conseq:<expr>
!      else alt:<expr> )
! l (block) ->
!     [let bndvar:<chars> = bndval:<expr>
!      body:<expr> ]
! f (fundef) ->
!     [func funname:<chars>  funformal:<chars> = funbody:<expr>
!      funscope:<chars> ]
! else (var) -> ident:<chars>

CrVar {factor. "#". "con". [["litval". chars]]};
CrVar {factor. "c". "call". [["rator". chars]. " ". ["rand". factor]]};
CrVar {factor. "(". "par". ["(". ["exp". expr]. ")"]};
CrVar {factor. "i". "conex".
  [NL + "(if ". ["cond". expr].
   NL + " then ". ["conseq". expr].
   NL + " else ". ["alt". expr]. " )"]};
CrVar {factor. "l". "block".
  [NL + "[let ". ["bndvar". chars]. " = ". ["bndval". expr].
   NL. ["body". expr]. " ]"]};
CrVar {factor. "f". "fundef".
  [NL + "[func ". ["funname". chars]. " ". ["funformal". chars]. " = ". ["funbody". expr].
   NL. ["funscope". expr]. " ]"]};
```

```
CrDef {factor, "var", [["ident", chars]]}.

PrNT {factor};
PrVariant {factor, "#"};
PrVariant {factor, "c"};
PrVariant {factor, "("};
PrVariant {factor, "i"};
PrVariant {factor, "l"};
PrVariant {factor, "f"};
PrDef {factor}.

displayn {"Test Grammar loaded."}.

define {root, "test_script",
 ["begin", "f", " ", "fac", "next", " ", "n", "next",
  "i", "=", " ", "n", "next", "#", " ", "0",
  "next", "#", " ", "1", "next",
  "*", " ", "n", "next", "c", " ", "fac", "next",
  "-", "(", "-", " ", "n", "next", "#", " ", "1", "next",
  "l", " ", "alpha", "next", "*", "#", " ", "512", "next", "#", " ", "3",
  "next", "c", " ", "fac", "next", " ", "alpha", "root"]}.
```

APPENDIX D: Transcript of Ω Session

The following is a transcript of an Ω session illustrating the operation of the prototype universal syntax directed editor and utilities presented in Appendices A and B, operating on the sample transaction grammar of Appendix C. The assertion 'Script {test_script}' causes the commands in testscript to be executed in order. Each command is printed on a separate line, followed by whatever output is generated by the programming environment. This transcript was produced by the McArthur interpreter [McArthur84].

```
% omega

OMEGA-1   11/30/84

Use Cntl-D or exit{} to quit.

For help, enter help{"?"}.

To report a bug, enter Bugs{}.

newrelation rule activated.

doit/redo activated: use 'doit' instead of 'do'.

> do {"PI6.USDE"}.

Universal SDE loaded.

OK

> do {"PI6.util"}.

PI-6 Utilities activated.

OK

> do {"PI6.testgram"}.

program:

else (prog) -> *:<expr>.

expr:

'=' (eql) -> left:<arex> ' = ' right:<arex>.

'n' (neq) -> left:<arex> ' <> ' right:<arex>.
```

```
else <arex> -> '' *:<arex> ''.

arex:

'+' (sum) -> left:<arex> ' + ' right:<term>.

'-' (dif) -> left:<arex> ' - ' right:<term>.

else <term> -> '' *:<term> ''.

term:

'*' (prd) -> left:<term> ' x ' right:<factor>.

'/' (quo) -> left:<term> ' / ' right:<factor>.

else <factor> -> '' *:<factor> ''.

factor:

'#' (con) -> litval:<chars>

'c' (call) -> rator:<chars> ' ' rand:<factor>.

'(' (par) -> '(' exp:<expr> ')'.

'i' (conex) -> '

(if ' cond:<expr> '

 then ' conseq:<expr> '

 else ' alt:<expr> ' )'.

'l' (block) -> '

[let ' bndvar:<chars> ' = ' bndval:<expr> '

 f' (fundef) -> '

[func ' funname:<chars> ' ' funformal:<chars> ' = ' funbody:<expr> '

else (var) -> ident:<chars>.

Test Grammar loaded.

OK

> Script {test_script}.
```

```
 begin
```

```
{<expr>}
```

```
f

[func {<chars>} <chars> = <expr>

<expr> ]
```

```
_fac

[func {fac} <chars> = <expr>

<expr> ]
```

```
next

[func fac {<chars>} = <expr>

<expr> ]
```

```
_n

[func fac {n} = <expr>

<expr> ]
```

```
next

[func fac n = {<expr>}

<expr> ]
```

```
i

[func fac n =

(if {<expr>}

 then <expr>
else <expr> )
<expr> ]
```

```
=

[func fac n =
(if {<arex>} = <arex>
 then <expr>
 else <expr> )
<expr> ]
```

```
_n

[func fac n =
(if {<chars>} = <arex>
 then <expr>
 else <expr> )
<expr> ]

[func fac n =
(if {n} = <arex>
 then <expr>
 else <expr> )
<expr> ]
```

```
 next

(if n = {<arex>}
 then <expr>
 else <expr> )
```

```
#

(if n = {<chars>}
 then <expr>
 else <expr> )
```

```
_0

(if n = {0}
 then <expr>
 else <expr> )
```

```
next

[func fac n =
(if n = 0
 then {<expr>}
 else <expr> )
<expr> ]
```

```
#

[func fac n =
(if n = 0
 then {<chars>}
 else <expr> )
<expr> ]
```

```
_1

[func fac n =
(if n = 0
 then {1}
 else <expr> )
<expr> ]
```

```
 next

[func fac n =
(if n = 0
 then 1
 else {<expr>} )
<expr> ]
```

```
 *

[func fac n =
(if n = 0
 then 1
 else {<term>} x <factor> )
<expr> ]
```

```
 _n

[func fac n =
(if n = 0
 then 1
 else {<chars>} x <factor> )
<expr> ]

[func fac n =
(if n = 0
then 1
else {n} x <factor> )
<expr> ]
```

```
next

(if n = 0
 then 1
 else n x {<factor>} )
```

```
c

(if n = 0
 then 1
 else n x {<chars>} <factor> )
```

```
_ fac

(if n = 0
 then 1
 else n x {fac} <factor> )
```

```
next
n x fac {<factor>}
```

```
_
n x fac {<chars>}
n x fac {<factor>}
Illegal Key: _
```

```
(

n x fac ({<expr>})
```

```
-

n x fac ({<arex>} - <term>)
```

```
_n

n x fac ({<chars>} - <term>)

n x fac ({n} - <term>)
```

```
next

(n - {<term>})
```

```
#

(n - {<chars>})
```

```
_1

(n - {1})
```

```
next

[func fac n =

(if n = 0

 then 1

 else n x fac (...) )

{<expr>} ]
```

```
I

[func fac n =
(if n = 0
 then 1
 else n x fac (...) )

[let {<chars>} = <expr>
<expr> ] ]
```

```
 _alpha

[func fac n =
(if n = 0
 then 1
 else n x fac (...) )

[let {alpha} = <expr>
<expr> ] ]
```

```
 next

[func fac n =
(if n = 0
 then 1
 else n x fac (... - ...) )

[let alpha = {<expr>}
<expr> ] ]
```

```
 *

[func fac n =
(if n = 0
 then 1
else n x fac (... - ...) )

[let alpha = {<term>} x <factor>

<expr> ] ]
```

```
#

[func fac n =

(if n = 0

 then 1

 else n x fac (... - ...) )

[let alpha = {<chars>} x <factor>

<expr> ] ]
```

```
_512

[func fac n =

(if n = 0

 then 1

 else n x fac (... - ...) )

[let alpha = {512} x <factor>

<expr> ] ]
```

```
next

[let alpha = 512 x {<factor>}

<expr> ]
```

```
#
```

```
[let alpha = 512 x {<chars>}
<expr> ]
```

```
_3

[let alpha = 512 x {3}
<expr> ]
```

```
 next

[func fac n =
(if n = 0
 then 1
 else n x fac (... - ...) )
[let alpha = 512 x 3
{<expr>} ] ]
```

```
 c

[func fac n =
(if n = 0
 then 1
 else n x fac (... - ...) )
[let alpha = 512 x 3
{<chars>} <factor> ] ]
```

```
 _fac

[func fac n =
(if n = 0
then 1
else n x fac (... - ...) )

[let alpha = 512 x 3
{fac} <factor> ] ]
```

```
next

[let alpha = 512 x 3
fac {<factor>} ]
```

```
_alpha

[let alpha = 512 x 3
fac {<chars>} ]

[let alpha = 512 x 3
fac {alpha} ]
```

```
root
{
[func fac n =
(if n = 0
 then 1
 else n x fac (...) )

[let alpha = 512 x 3
fac alpha ] ]}
Done
> ~DepthLimit(5). DepthLimit(20).
20
```

```
> Command {"display"}.

{

[func fac n =

(if n = 0

 then 1

 else n x fac (n - 1) )


[let alpha = 512 x 3

fac alpha ] ]}

OK

> exit {}.

Goodbye.
```

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Office of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93943	40
Associate Professor Bruce J. MacLennan Department of Computer Science Ayres Hall The University of Tennessee Knoxville, TN 37996-1301	12
Dr. Ralph Wachter Code 433 Office of Naval Research 800 N. Quincy Arlington, VA 22217-5000	1
Dr. Stephen Squires DARPA Information Processing Techniques Office 1400 Wilson Boulevard Arlington, VA 22209	1
Professor Jack M. Wozencraft, 62Wz Department of Electrical and Comp. Engr. Naval Postgraduate School Monterey, CA 93943	1
Dr. Joseph Goguen DN-154 SRI International 333 Ravenswood Avenue Menlo Park, CA 94025	1
Professor Rudolf Bayer Institut für Informatik Technische Universität Postfach 202420 D-8000 Munchen 2 West Germany	1

Dr. Robert M. Balzer
USC Information Sciences Inst.
4676 Admiralty Way
Suite 10001
Marina del Rey, CA 90291 1

Mr. Ronald E. Joy
Honeywell, Inc.
Computer Sciences Center
10701 Lyndale Avenue South
Bloomington, MI 55402 1

Mr. Ron Laborde
INMOS
Whitefriars
Lewins Mead
Bristol
Great Britain 1

Mr. Lynwood Sutton
Code 424, Building 600
Naval Ocean Systems Center
San Diego, CA 92152 1

Mr. Jeffrey Dean
Advanced Information and Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, CA 94040 1

Mr. Jack Fried
Mail Station D01/31T
Grumman Aerospace Corporation
Bethpage, NY 11714 1

Mr. Dennis Hall
New York Videotext
104 Fifth Avenue, Second Floor
New York, NY 10011 1

Professor S. Ceri
Laboratorio di Calcolatori
Departimento di Elettronica
Politecnico di Milano
20133 - Milano
Italy 1

Mr. A. Dain Samples
Computer Science Division - EECS
University of California at Berkeley
Berkeley, CA 94720 1

Antonio Corradi
Dipartimento di Elettronica
Informatica e Sistemistica
Universita Degli Studi di Bologna
Viale Risorgimento, 2
Bologna

Italy 1

Dr. Peter J. Welcher
Mathematics Dept., Stop 9E
U.S. Naval Academy
Annapolis, MD 21402 1

Dr. John Goodenough
Wang Institute
Tyng Road
Tyngsboro, MA 01879 1

Professor Richard N. Taylor
Computer Science Department
University of California at Irvine
Irvine, CA 92717 1

Dr. Mayer Schwartz
Computer Research Laboratory
MS 50-662
Tektronix, Inc.
Post Office Box 500
Beaverton, OR 97077 1

Professor Lori A. Clarke
Computer and Information Sciences Department
LGRES ROOM A305
University of Massachusetts
Amherst, MA 01003 1

Professor Peter Henderson
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794 1

Dr. Mark Moriconi
SRI International
333 Ravenswood Avenue
Manlo Park, CA 95025 1

Professor William Waite
Department of Electrical and Computer Engineering
The University of Colorado
Campus Box 425
Boulder, CO 80309-0425 1

Professor Mary Shaw
Software Engineering Institute
Carnegie-Mellon University
Pittsburgh, PA 15213 1

Dr. Warren Teitelman
Engineering/Software
Sun Microsystems Federal, Inc.
2550 Garcia Avenue
Mountain View, CA 94031 1

Prof. Raghu Ramakrishnan **Univ. of Texas at Austin** **Dept. of Computer Science** **Austin, TX 79712**	1
Mr. Hyoung Joo Kim 3567-A, Lake Austin Blvd Austin, TX 78703	1
Ms. Suzana Hutz Computer Language Lab, MS 47LH Hewlett Packard 19447 Pruneridge Ave Cupertino, CA 95014	1
Vinay Apsingikar CMC Limited 115 Sarojini Devi Road Secunderabad 500003 India	1
Keith Kramer Human Performance Laboratory The Catholic University of America Washington, D.C. 20064	1